Management Science Research Report No. MSRR-596

A PARALLEL IMPLEMENTATION OF THE COLUMN SUBTRACTION ALGORITHM

T. H. C. Smith[*]

Gerald L. Thompson[**]

December 22, 1993

[*] Department of Computer Science
Rand Afrikaans University, P.O. Box 524
Aucklandpark 2006, South Africa

[**] Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213

The research underlying this report was supported by the Pittsburgh Supercomputing Center and the U. S. Office of Naval Research under Contract No. N00014-85-K-0198 NR 047-048.

Management Sciences Research Group
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract : We have implemented Harche and Thompson's column subtraction algorithm for the set partitioning problem on a CM-200 Connection Machine. The implementation involved partitioning the large array of processors in the CM-2 into segments and letting each segment explore a different part of the search tree generated by the column subtraction algorithm. Our reported computational results indicate that the segments are highly utilized and that good speedups are obtained as the number of segments is increased.

Keywords : Parallel processing; branch-and-bound method; column subtraction; set partitioning

1 Introduction

Parallel computers make it possible to speed up the solution of a problem by simultaneously using more than one processor in solving the problem. One solution method that lends itself naturally to parallelization is a branch-and-bound algorithm (which solves a large problem by first partitioning it into

[*]The computational experiments were performed while Smith was a Visiting Scholar at the Graduate School of Industrial Administration, Carnegie-Mellon University.

[†]Research supported by the Pittsburgh Supercomputing Center and the U.S. Office of Naval Research under Contract No. N00014-85-K-0198 NR 047-048.

subproblems and then solving each of the subproblems). In a parallel branch-and-bound algorithm the different subproblems can be solved simultaneously (in parallel) by different processors.

In an asynchronous parallel branch-and-bound algorithm each processor independently obtains a subproblem from a list of subproblems and solves it (possibly inserting new subproblems into the list). This continues until the list is empty and all processors are idle.

Most of the experimental results for asynchonous parallel branch-and-bound algorithms reported in the literature were obtained using a relatively small number of processors. An example of such research is the work done by Rushmeier and Nemhauser [4] who used a machine with 45 processors.

In [2] Loots and Smith experimented with an asynchronous parallel branch-and-bound algorithm for the 0-1 knapsack problem using a machine with only 8 processors. In their algorithm different parts of the search tree are assigned to different processors. Each of these processors then asynchronously performs a depth first search on the part of the search tree assigned to it.

In this paper we will investigate a synchronous parallel branch-and-bound algorithm using thousands of processors. The array of processors is partitioned into segments, the problem is divided into the same number of subproblems as the number of segments and each segment synchronously perform the same depth first branch-and-bound algorithm on a different subproblem.

2 The column subtraction algorithm

Let A be an m x n binary matrix, e a column vector of m ones, c a vector of n positive integer costs and x a column vector of n decision variables. The set partitioning problem (SPT) may then be defined as

$$\begin{array}{lrcl} \text{minimize} & cx & & \\ \text{subject to} & Ax & = & e \\ & x & \in & \{0,1\} \end{array}$$

Suppose T is an $(m+1)$ x $(k+1)$ matrix containing an optimal condensed simplex tableau for the linear programming relaxation of SPT (note that $k = n - m$). The columns of T are identified by the superscripts $0, \ldots, k$ and the elements of a column by the subscripts $0, \ldots, m$. Column T^0 contains the negative of the optimal objective function value in T_0^0 and the values of the basic variables in its other elements. For $i > 0$ column T^i contains the reduced cost of a nonbasic variable in T_0^i and the basis representaion of the corresponding A column in its other elements. It is assumed that columns

$T^1, \ldots, T^k$ are in nondecreasing order of reduced costs.

The column subtraction algorithm of Harche and Thompson [1] for the SPT performs a depth first search and is formulated in a pseudo-C language in Figure 1. In the algorithm *ub* is an upper bound on the objective function value of an optimal set partition. E is an $(m+1)$ x $(k+1)$ matrix used to record the solutions at the various nodes of the depth first search tree and *save* is a vector used to record which columns of T have been subtracted to get to the current node in the search tree.

The initial value of *tlim* acts as an upper bound on the distance in T between the first and last columns subtracted. As such it limits the depth of the search and makes the algorithm an effective heuristic which finds good set partitions quickly. The algorithm can be changed into an exact algorithm by initializing *tlim* to k and deleting the if statement just before the end of the outer loop.

The second condition in the while loop allows a forward move to be made only if such a move will lead to a *live* node, *i.e.* a node for which the lower bound on the optimal cost of a set partition is less than the upper bound.

```
$t$ = 1; /* index of next column to be subtracted */
$e$ = 0; /* number of columns subtracted for current solution */
$E^0$ = $T^0$;
initialize $ub$ and $tlim$;
do {
    while(($t$ <= $tlim$) && ($T_0^t$ - $E_0^e$ < $ub$)) {
        $e$++; /* move forward in search tree */
        $E^e$ = $E^{e-1}$ - $T^t$;
        $save_e$ = $t$;
        if($E^e$ defines a partition) $ub$ = $E_0^e$;
    }
    $t = save_e$ + 1; e--; /* backtrack */
    if(($e$ == 0) && ($t$ < $tlim$)) $tlim$++;
} while($e$ >= 0);
```

Figure 1: Column subtraction algorithm.

3 Parallelization

If the column subtraction algorithm is executed by a single processor the $m+1$ subtractions involved in a column subtraction (the second step in the while loop of the algorithm) must be carried as $m+1$ sequential operations. On a multiprocessor these subtractions can be done in parallel by $m+1$ different processors. Also the test for a partition can be parallelized by simultaneously testing the last m elements of E^e for integrality and nonnegativity on m different processors.

We used a CM-2 Connection Machine with 32K processors in our study. Each of these processors has its own 32KB local memory. The column subtraction algorithm can be parallelized by storing each row of the matrices T and E in the local memory of a different processor and letting each processor handle all subtractions and tests for the row in its memory.

However, on the CM-2 the number of processors allocated to a program is always a multiple (a power of 2) of the number of processors in a quadrant (8K). If m, the number of rows in A, is much smaller than 8K then the processor utilization will be very low if the column subtraction algorithm is parallelized as indicated in the previous paragraph.

In order to obtain better processor utilization we logically partition the array of physical processors into segments where the size of a segment is a power of 2 which exceeds m. Since the number of physical processors is a power of 2, it follows that the number of segments is also a power of 2, say 2^s. We identify the segments by the numbers $0, \ldots, 2^s - 1$.

The matrix T is initially copied to all segments (one row per processor for the first $m + 1$ processors in each segment). Each segment then executes the column subtraction algorithm on a different part of the total search tree using its own E matrix. Segment i starts out by logically setting the first s nonbasic variables equal to the bits in the s-bit binary representation of the segment number i. This is done by initializing E^0 to T^0 and then subtracting the combination of the first s columns of T corresponding to the nonzero bits in the binary representation of i. For example if $s = 3$ then segment 5 will compute $E^0 = T^0 - T^1 - T^3$ since the 3-bit binary representation of 5 is 101_2.

After initializing E^0 each processor simultaneously tests whether E^0 defines a partition. Also the index t is initialized to $s + 1$ to take into account that the first s columns have already been considered when the segments simultaneously enter the main loop of the column subtraction algorithm.

In the CM-2 each processor executes the same instruction sequence in lockstep. This implies that each segment has to continue moving forward in its part of the search tree so long as at least one of the segments is able to move forward to a live node. All the segments backtrack simultaneously only when no segment is able to move forward to a live node. Since some segments may be making useless forward moves to dead nodes, it is of interest to observe the *segment utilization* which is calculated at the end of the computation as the ratio between the total number of live nodes explored and the total number of explored nodes.

We implemented the parallelized column subtraction algorithm in the C* language [5]. More detail of this implementation is provided in the appendix which assumes that the reader has a working knowledge of C*.

4 Computational experience

We tested the parallelized column subtraction algorithm on the CM-2 Connection Machine at the Pittsburgh Supercomputing Center. Our computational experience was obtained with a set of 5 random set partitioning problems and a set of 5 three index assignment problems. Because of a limit on the computing time available to us on the CM-2 we solved the linear programming relaxation of the SPT on a different machine and transferred the optimal tableau T to the CM-2.

The random set partioning problems each have 50 rows and 1000 columns. They were randomly generated with an average density of 3.26% having at least one nonzero in each column and at least two nonzero entries in each row. The costs were generated randomly in the interval [1,100]. These problems were solved using 1, 2, 4, ..., 64 and 128 segments. In all problems we started the algorithm with $tlim = 15$ and ub 20% above the lower bound T_0^0. In Table 1 we report the averages for the number of live nodes explored, the run time (in seconds), the segment utilization and the speedup (relative to the run time using only one segment). Figure 2 is a line graph of the average speedups for the different number of segments.

A three index assignment problem is an SPT with a special structure. It has $3n$ rows and n^3 columns where n is a positive integer (see [1] for more detail). We considered five such problems with $n = 15$ and the costs randomly generated in the interval [1,100]. Since these problems have a density of 6.67% the column subtraction algorithm takes longer to solve them than for the random set partitioning problems. For this reason these problems were only solved using 16, 32, 64 and 128 segments. In all these problems we

Number of segments	Number of live nodes	Run time	Segment utilization	Speedup
1	134971.8	258.6	1.000	
2	144111.0	138.9	0.991	1.81
4	150365.2	72.4	0.986	3.51
8	145460.4	35.0	0.981	7.27
16	171677.2	21.0	0.977	12.66
32	180887.6	11.3	0.958	22.56
64	157819.0	5.0	0.950	50.06
128	264844.6	4.4	0.933	83.06

Table 1: Averages for random set partitioning problems.

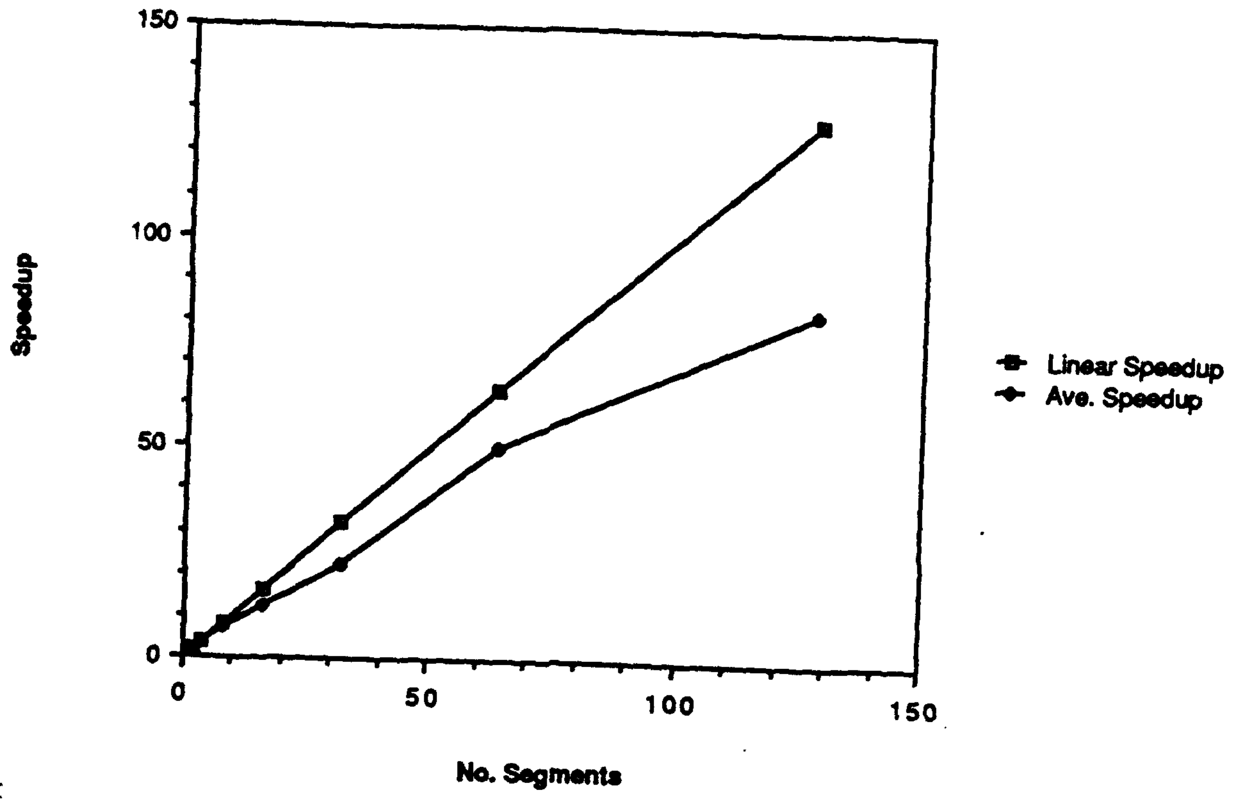

Figure 2: Average speedup for random set partitioning problems.

Number of segments	Number of live nodes	Run time	Segment utilization	Speedup
16	3328837.6	441.2	0.950	
32	3275352.0	221.8	0.933	2.07
64	3186272.2	112.8	0.907	4.58
128	3115957.6	58.1	0.879	12.46

Table 2: Averages for three index assignment problems.

started the algorithm with *tlim* = 20 and *ub* 30% above the lower bound T_0^0. In Table 2 we again report the average performance (with the speedup relative to the run time using 16 segments). Figure 3 is a line graph of the average speedups for the different number of segments.

The large average segment utilization observed for both problem sets can be explained by the fact that the reduced costs $T_0^1, \ldots, T_0^s$ of the first s nonbasic variables are often very close to zero with the result that E_0^e will (almost) be the same for all segments and any e. This means that most of the time all segments are able to move forward to live nodes, resulting in a large segment utilization. Of course, this becomes less true as s increases (as

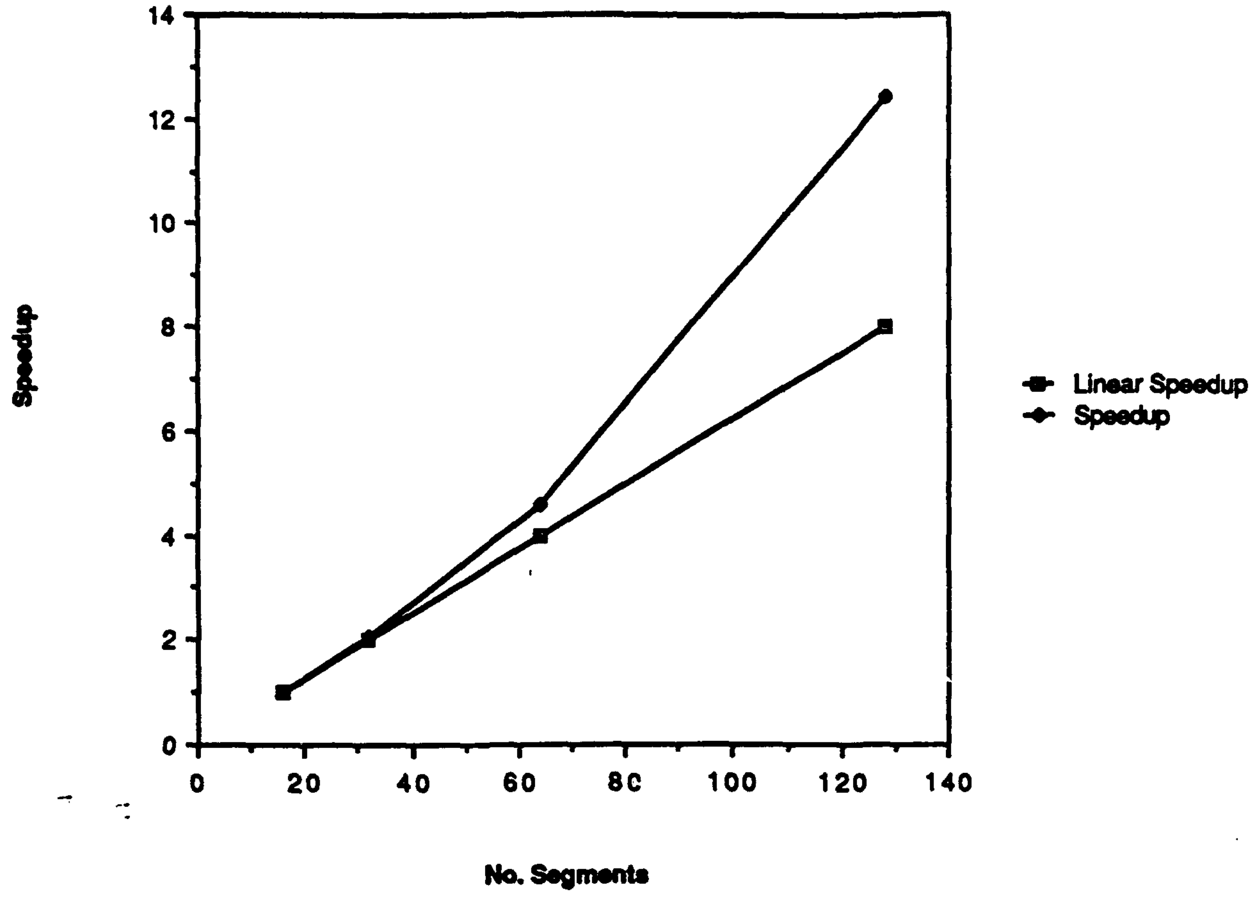

Figure 3: Average speedup for three index assignment problems.

reflected in the two tables by the decreasing average segment utilization for increasing number of segments).

As can be seen from the results in Table 2 it is possible to obtain super-linear speedup, *i.e.* speedups of more than the factor by which the number of segments increase. This possibility has been pointed out by Lai and Sahni [3].

5 Conclusion

We have implemented the column subtraction algorithm for the set partitioning problem on a CM-2 Connection Machine. To the best of our knowledge this is the first implementation of a synchronous branch-and-bound algorithm for a massively parallel computer. One of the problems in implementing an algorithm for such a computer is ensuring that all or most of the processors are utilized throughout the computation.

Our implementation involved partitioning the large number of processors into segments and letting each segment synchronously explore a different part of the search tree generated by the column subtraction algorithm. The reported computational results indicate that the different segments are highly utilized and that good speedups are obtained as the number of segments is increased.

The column subtraction algorithm is also applicable to other combinatorial optimization problems such as set packing and set covering. Our limited computational experience with a parallelized column subtraction algorithm for the set covering problem indicate that in that case high segment utilization and good speedups are also obtainable.

Appendix

The program uses the following global variables:

```
bool:physical zero_row, /* true if processor handles row 0 */
 basic_row, /* true if processor handles one of rows 1 to m */
 scansets, /* defines scan set for each segment */
 flag; /* test result */
unsigned char:physical seg_num; /* segment number */
short:physical row_num; /* number of row handled */
float:physical nub; /* negative of current upper bound */
int seg_size, /* number of processors in a segment */
    seg_count, /* number of segments */
    log_seg_count; /* logarithm base 2 of seg_count */
float epsilon = 0.01f;
```

The parallel variables are initialized as follows:

```
with(physical) {
  nub = -ub; /* assume upper bound ub has been initialized */
  row_num = pcoord(0) % seg_size;
  zero_row = (row_num == 0);

  basic_row = (!zero_row) & (row_num <= m);

  scansets = !basic_row;

  seg_num = pcoord(0) / seg_size;

}
```

We next list three functions used in the program. The first column of E is set up for all segments before entering the do loop in the column subtraction algorithm by calling the **setup_e** function. The condition in the while statement uses the function LB_lt_UB function to determine the number of segments which are able to make a forward move to a live node. If it returns 0 then all the segments leave the while loop and backtrack in parallel. Otherwise all the segments move forward to new nodes in parallel. The **test_solutions** function is used within the while loop after a forward move to test if any of the segments have found a set partition which yields a smaller upper bound on the optimal cost of a set partition.

```
void setup_e(float:physical *e, float:physical *t) {
/* e and t point to the first columns of E and T */
  unsigned char:physical work;
  *e = *t; /* copy first T column into first E column */

  work = seg_num;

  for(int i=1; i<=log_seg_count; i++) {

    where(work % 2) *e -= *(t+i);

    work >>= 1;

} }


int LB_lt_UB(float:physical *e,float:physical *t) {

/* e points to current column of E, t to next column of T */

  where(zero_row & (floor(*e - *t + epsilon) > nub))

    return(+= (int:physical) 1);

}


int test_solutions(float:physical *e) {

/* e points to current column of E */

/* if(smaller ub found) return(1); else return(0); */

  float max_nub;
```

```
/* nonnegativity test */
where(basic_row) flag = (*e > -epsilon);
flag = scan(flag,0,CMC_combiner_logand,CMC_downward,
            CMC_segment_bit,&scansets,CMC_exclusive);
if(|= (zero_row & flag)) {
  /* integrality test */
  where(basic_row) flag = (*e * (1.0f - *e) < epsilon);
  flag = scan(flag,0,CMC_combiner_logand,CMC_downward,
              CMC_segment_bit,&scansets,CMC_inclusive);
  where(zero_row) {
    flag &= (nub < floor(*e + epsilon));
    if(|= flag) {
      /* at least one segment found a better partition */
      where(flag) {
        nub = floor(*e + epsilon);
        max_nub = >?= nub;
      }
      nub = max_nub; /* distribute new nub */
      return(1);
  } } }
  return(0);
}
```

References

1. F. Harche and G.L. Thompson, "The column subtraction algorithm: an exact method for solving weighted set covering, packing and partitioning problems", forthcoming in *Computers and Operations Research* (1994).

2. W. Loots and T.H.C. Smith, "A parallel algorithm for the 0-1 knapsack problem", *International Journal of Parallel Programming* 21 (1992) 349-362.

3. T. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms", *Journal of the ACM* 27 (1984) 594-602.

4. R.A. Rushmeier and G.L. Nemhauser, "Experiments with parallel branch-and-bound algorithms for the set covering problem" *Operations Research Letters* 13 (1993) 277-285.

5. Thinking Machines Corporation, *C* Programming Guide* , Cambridge, Massachusetts, 1993.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MSRR-596	2. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A PARALLEL IMPLEMENTATION OF THE COLUMN SUBTRACTION ALGORITHM		5. TYPE OF REPORT & PERIOD COVERED Technical Report, Dec. 1993
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(S) T.H. C. Smith Gerald L. Thompson		8. CONTRACT OR GRANT NUMBER(S) N00014-85-K-0198
9. PERFORMING ORGANIZATION NAME AND ADDRESS Graduate School of Industrial Administgration Carnegie Mellon University Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (Code 434) Arlington, VA 22217		12. REPORT DATE December ±993
		13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)		15. SECURITY CLASS (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Parallel processing Branch-and-Bound method Column subtraction Set partitioning		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We have implemented Harche and Thompson's column subtraction algorithm for the set partitioning problem on the CN-200 Connection Machine. The implementation involved partitioning the large array of processors in the CM-2 into segments and letting each segment explore a different part of the search tree generated by the column subtraction algorithm. Our reported computational results indicate that the segments are highly utilized and that good speedups are obtained as the number of segments is increased.